— Lecture Notes —

Left Corner Parsing

1 The Left Corner Strategy

1 Top down parsers suffer from the problem of left recursion, but even if you eliminate that they still suffer from the problem of not making contact with the data until after they've done a lot of predicting, maybe wrongly. Bottom up parsers are more data driven: they focus on what is actually there in the input, but they are unable to use context to constrain how they analyze the data they are currently looking at. Even here, it may be a very long time before a wrong analysis is recognized to be false, and backtracking to the place where the wrong step was actually made can take a very long time.

2 Left corner parsers combine a bottom up data driven component with a top down predictive component, and so achieve what is in very many cases the best of both worlds.

The fundamental idea is as follows. Given any grammar rule of the form $A \longrightarrow B\ C_1 \ldots C_n$, call B *the left corner of* A. We then take the reflexive transitive closure of this relation, which we also call the *left corner relation*. So, given a sentence like *the villain heard the hero sneeze*, the first occurrence of the word *the* is a left corner of the matrix S. So is the NP *the villain*. And, because the relation is reflexive, so is the matrix S a left corner of itself.

The left corner parsing strategy uses the left corner relation in the following way. In order to prove that the input sentence belongs to category S, we try to show first that the first word of the sentence is a left corner of S. Because of the recursive nature of the relation (because it is a closure), this involves finding a rule of which *the* is the left corner, for example, a rule like $Det \longrightarrow the$. Then we continue, attempting to show now that Det is a left corner of S, and so on.

There's one hitch: Suppose that we have a rule like $NP \longrightarrow DetN$, of which Det is the left corner. We don't just blindly go ahead and see if NP can also be a left corner of S. First we pause and make sure that the next bit of string can be parsed as an N. So: we know we might have a Det, based on looking at the data. We predict that we have an N, and then try to verify that prediction. (In the same way that we are trying to verify that the whole string is an S: by using the left corner strategy.)

In this way, the left corner strategy mixes bottom up parsing and top down parsing: go up from the bottom until you hit a branching node, then use a prediction to guide the next phase of parsing.

3 **Empty Productions** This version can be extended for use with grammars in which we allow ϵ-productions: instead of grabbing the first word in a sentence, we define a `leaf` as either the first word of a sentence or the empty string `[]`.

```
rec( String ):-
        lc_rec( s, String, [] ).

lc_rec( C, L0, L ):-
        L0 = [W|L1],
        word( W, C_W ),
        lc( C_W, C, L1, L ).

%%% C1 is a left corner of C2 if
%%% (1) There is a rule C3 --> C1, Rest.
%%% (2) The unprocessed prefix of the string can be
%%%     recognized as Rest.
%%% (3) C3 is a left corner of C2.
%%% [Base Case: Everything is a left corner of itself.]

lc( C, C, L, L ).
lc( C1, C2, L0, L ):-
        rule( C3, [C1|RestRHS] ),  % rule(s,[np,vp])
        lc_star( RestRHS, L0, L1 ),
        lc( C3, C2, L1, L ).

lc_star( [], L, L ).
lc_star( [C1|RestCs], L0, L ):-
        lc_rec( C1, L0, L1 ),
        lc_star( RestCs, L1, L ).
```

Figure 1: A Prolog left corner recognizer.

Then we ask "Is there a leaf (at the left edge of the remaining words) that is a left corner of the category we are trying to recognize?"

4 **Oracles** Suppose the first word of the sentence can be assigned a category which is never the left corner of an S. (I suppose for example that the word *hits*, which can be analyzed as a V, should never be so analyzed when it is the first word of the sentence, but there may be counter-examples for this specific case.) We would like to know that as soon as possible, so as to backtrack quickly and try something else.

The question of who can be a left corner of whom is actually independent of any particular input. It depends only on the grammar itself. So the left corner relation can be computed *off line*, and stored as a table. Then, after calling either `word/2` or `rule/2`, we immediately ask if the thing we just looked up (the category of a word or the mother node in a rule) can be a left corner of the category we are trying to parse. This is now just a matter of table lookup, a very fast operation.

This kind of meta-information about the grammar is an example of a *parsing oracle*. In a sense the left corner link table allows you to "see into the future" and discover if this parse in fact *has* a future.

5 Before you launch into the rest of this stuff, note first that any Prolog program which uses difference lists can be rewritten in the DCG formalism. There is nothing about the nature of the DCG rule compiler that requires the rules it compiles to be about grammar! Consider for example the following definition of the `yields` predicate, that returns the list of leaf labels from a tree.

```
yields( -Leaf, [Leaf|L], L ).
yields( Root/SubTrees, L0, L ):-
        yields_star( SubTrees, L0, L ).

yields_star( [], L, L ).
yields_star( [T1|RestSubs], L0, L ):-
        yields( T1, L0, L1 ),
        yields_star( RestSubs, L1, L ).
```

This can be rewritten as a DCG as follows.

```
yields( -Leaf ) --> [Leaf].
yields( Root/SubTrees ) -->
        yields_star( SubTrees ).

yields_star( [] ) --> [].
yields_star( [T1|RestSubs] ) -->
        yields( T1 ),
        yields_star( RestSubs ).
```

2 Applying Left Recursion Removal to a Left Corner Recognizer

6 Background Reading Take a look at Dale Gerdemann's "Parsing as Tree Traversal." (*Proceedings of COLING 94*, pp. 396–400.) It can be found on the course web page. Gerdemann observes that the three parsing algorithms that we have looked at so far (top down, bottom up and left corner) correspond to the standard traversal orders for trees. But thought of in this way, the programs for bottom up and left corner parsing are left recursive. The treatment in that paper of how to remove the left recursion from the parsing programs is a little complex, and refers in some places to literature you don't know (I presume). So these notes re-cover some of that territory at a slower pace.

In these notes we tackle the specific problem of getting a left corner recognizer from the form naturally suggested by an inorder traversal of a tree into a non-left-recursive form useable by a deductive engine like Prolog.

7 The basic program (Adapted from Gerdemann 1994.)

```
lc_rec(PreTermCat) -->
        [Word],
        {word(Word,PreTermCat)}.  % e.g., word(her,det)
lc_rec(Mother) -->
        lc_rec(Left),
        {rule(Mother,Left,Right)}, % e.g., rule(s,np,vp)
        lc_rec(Right).
```

We assume for the moment that all rules are binary branching.

8 This program is left recursive. We will attempt to use the techniques we covered before to transform it in the direction of Greibach Normal Form until we get rid of the left recursion.

9 Fortunately, this "grammar" has very few nonterminal categories to worry about (only `lc`!). We have one "alpha rule" ($A \longrightarrow A\alpha$):

```
lc_rec(Mother) -->
        lc_rec(Left),
        {rule(Mother,Left,Right)},
        lc_rec(Right).
```

and one "beta rule" ($A \longrightarrow \beta$):

```
lc_rec(PreTermCat) -->
        [Word],
        {word(Word,PreTermCat)}.
```

So:

α = {`rule(Mother,Left,Right)`},`lc_rec(Right)`
β = `[Word]`,{`word(Word,PreTermCat)`}

(Keep in mind that the left recursion is inherent in the left corner recognizer, not in the underlying linguistic grammar. We will be transforming the recognizer only. The clauses defining `word/2` and `rule/3` will not be affected.)

10 Let $B = \mathtt{b}$ be the fresh nonterminal that we introduce. Now we replace all the rules for $A = \mathtt{lc\_rec}$ with

$$\begin{array}{ll} A \longrightarrow \beta & B \longrightarrow \alpha \\ A \longrightarrow \beta B & B \longrightarrow \alpha B \end{array}$$

The Prolog code is in Fig. 2.

```
lc_rec(C) -->
        [Word],
        {word(Word,C)}.
lc_rec(C) -->
        [Word],
        {word(Word,C1)},
        b(C,C1).

b(C,C1) -->
        {rule(C,C1,Right)},
        lc_rec(Right).
b(C,C1) -->
        {rule(C2,C1,Right)},
        lc_rec(Right),
        b(C,C2).
```

Figure 2: The left corner recognizer without left recursion.

11 The Features The transformation towards GNF does not tell us what to do with the features of nonterminals. This is particularly tricky, since we don't even really know *how many* features to assign to our new category.

To solve this problem in general it is crucial to keep in mind how the proof trees for the transformed program relate to the proof trees for the untransformed program.

Recall from earlier coursenotes the graphical view of the transformation, repeated here in Fig. 3. The idea is that the first piece of string that we recognize

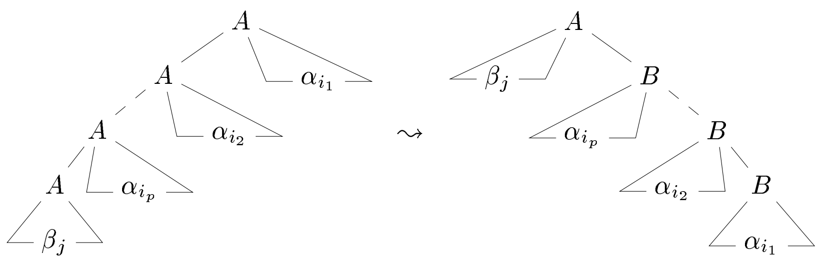

Figure 3: A left recursive derivation tree and the transformed tree.

(β_j in the figure) is "underlyingly" the left corner of the constituent dominating the prefix $\beta_j \alpha_{i_p}$ in the untransformed program. This holds true even when we are talking about arbitrary programs (like a recognizer program), and not about context free grammars themselves. The trees in this case should be thought of as proof trees rather than derivation trees, but we know already that these two concepts are tightly related.

So in general the structure assigned by the transformed program will be the left corner of a structure which will be built up as we go down the (recursive) right branch of the proof tree. In particular, any features we pick up in proving an instance of a `b` goal (and likewise any features we pick up in proving the initial `lc_rec` goal) will be needed "downstairs" in the subsequent recursive call to `b`. That means that the features of a given word picked up in a call to `word/2` and the features of the mother category picked up in a call to `rule/3` will need to be passed down to the recursive call to `b`. In this example the only features we are dealing with are major category features, but any more specific information about the categories we discover will need to be brought along as well in more sophisticated programs.

12 In Fig. 4 we see the intended proof tree in the original, untransformed program (which we will never get, because the infinite branch of the left recursive program gets in the way). In Fig. 5 we see (part of) the proof tree we actually get in the transformed program.

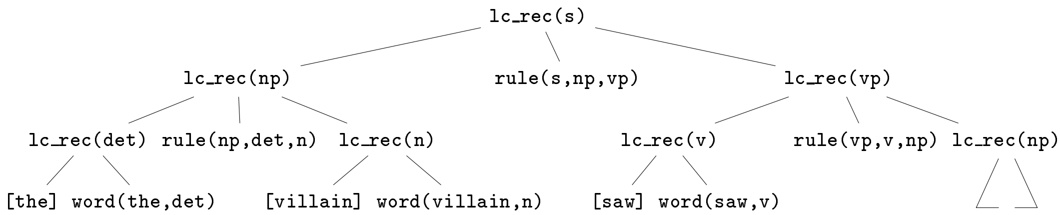

Figure 4: Proof tree for *the villain saw the hero.*

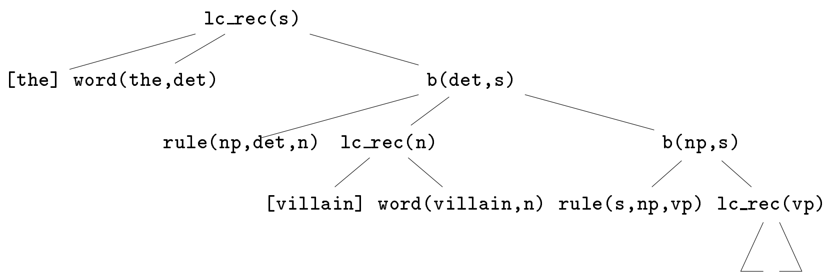

Figure 5: Actual proof tree in the transformed grammar.

13 Given these observations, we can see that the "mystery relation" `b` in fact expresses exactly the "is a left corner of" relation between categories of the grammar. (Actually, given the order of the arguments, this would be better read as "has as a left corner".)

14 A final simplification As Gerdemann notes, we can simplify our program further if we allow empty productions. Then the presence of the optional `b` can be expressed by a rule expanding `b` to `[]`. At the same time, we rename `b` to `lc`, since, as noted, it expresses the "is a left corner of" relation between its (first) two arguments.

```
lc_rec(C) -->
        [Word],
        {word(Word,C1)},
        lc(C,C1).

lc(C,C) --> [].
lc(C,C1) -->
        {rule(C2,C1,Right)},
        lc_rec(Right),
        lc(C,C2).
```

It should be noted that in this transformed form, and up to the limitation of our grammar to binary branching rules, this is essentially the same program given by Pereira and Shieber (1987) in their presentation of left corner parsing (pp. 180ff.).